

Lecture 7 – C Memory Management



2007-02-06

Hello to Said S. from
Columbus, OH

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

2x Phase-change memory! ⇒

Engineers are working on the non-volatile replacement for flash & DRAM memory, called “phase-change memory”. Intel and ST Microelectronics have just found a way to double the storage capacity without adding to cost of the fab process. This is the future!



Review

- **C has 3 pools of memory**
 - **Static storage**: global variable storage, basically permanent, entire program run
 - **The Stack**: local variable storage, parameters, return address
 - **The Heap** (dynamic storage): `malloc()` grabs space from here, `free()` returns it.
- **`malloc()` handles free space with freelist. Three different ways to find free space when given a request:**
 - **First fit** (find first one that's free)
 - **Next fit** (same as first, but remembers where left off)
 - **Best fit** (finds most “snug” free space)



Slab Allocator

- A different approach to memory management (used in GNU `libc`)
- Divide blocks in to “large” and “small” by picking an arbitrary threshold size. Blocks larger than this threshold are managed with a freelist (as before).
- For small blocks, allocate blocks in sizes that are powers of 2
 - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes

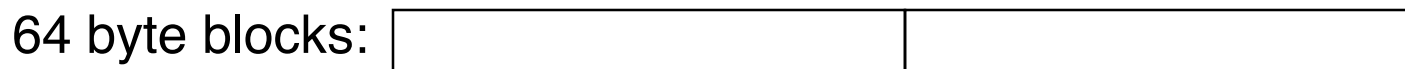
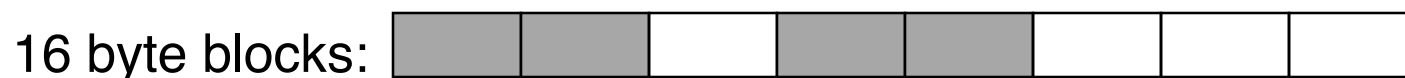


Slab Allocator

- **Bookkeeping for small blocks is relatively easy: just use a *bitmap* for each range of blocks of the same size**
- **Allocating is easy and fast: compute the size of the block to allocate and find a free bit in the corresponding bitmap.**
- **Freeing is also easy and fast: figure out which slab the address belongs to and clear the corresponding bit.**



Slab Allocator



16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00



Slab Allocator Tradeoffs

- **Extremely fast for small blocks.**
- **Slower for large blocks**
 - **But presumably the program will take more time to do something with a large block so the overhead is not as critical.**
- **Minimal space overhead**
- **No fragmentation (as we defined it before) for small blocks, but still have wasted space!**



Internal vs. External Fragmentation

- With the slab allocator, difference between requested size and next power of 2 is wasted
 - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- We also refer to this as fragmentation, but call it *internal fragmentation* since the wasted space is actually within an allocated block.
- **External fragmentation**: wasted space between allocated blocks.



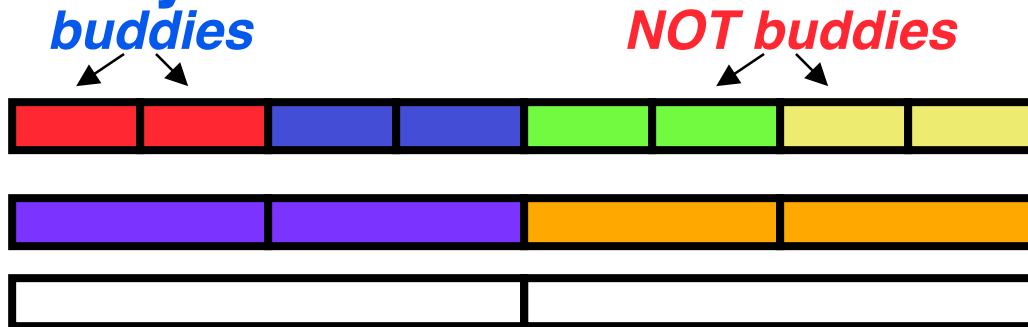
Buddy System

- **Yet another memory management technique (used in Linux kernel)**
- **Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)**
- **Keep separate free lists for each size**
 - **e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.**



Buddy System

- If no free block of size n is available, find a block of size $2n$ and split it in to two blocks of size n
- When a block of size n is freed, if its neighbor of size n is also free, combine the blocks in to a single block of size $2n$
- **Buddy** is block in other half larger block



- Same speed advantages as slab allocator



Allocation Schemes

- **So which memory management scheme (K&R, slab, buddy) is best?**
 - **There is no single best approach for every application.**
 - **Different applications have different allocation / deallocation patterns.**
 - **A scheme that works well for one application may work poorly for another application.**



Automatic Memory Management

- Dynamically allocated memory is difficult to track – why not track it **automatically**?
- If we can keep track of what memory is in use, we can reclaim everything else.
 - Unreachable memory is called **garbage**, the process of reclaiming it is called **garbage collection**.
- So how do we track what is in use?



Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
 - We can do this in constant space by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?



Tracking Memory Usage

- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
 - E.g., simple Lisp/Scheme system with only cons cells (61A's Scheme not “simple”)
- Could use a *strongly typed* language (e.g., Java)
 - Don't allow conversion (casting) between arbitrary types.
 - C/C++ are not strongly typed.
- Here are 3 schemes to collect garbage



Scheme 1: Reference Counting

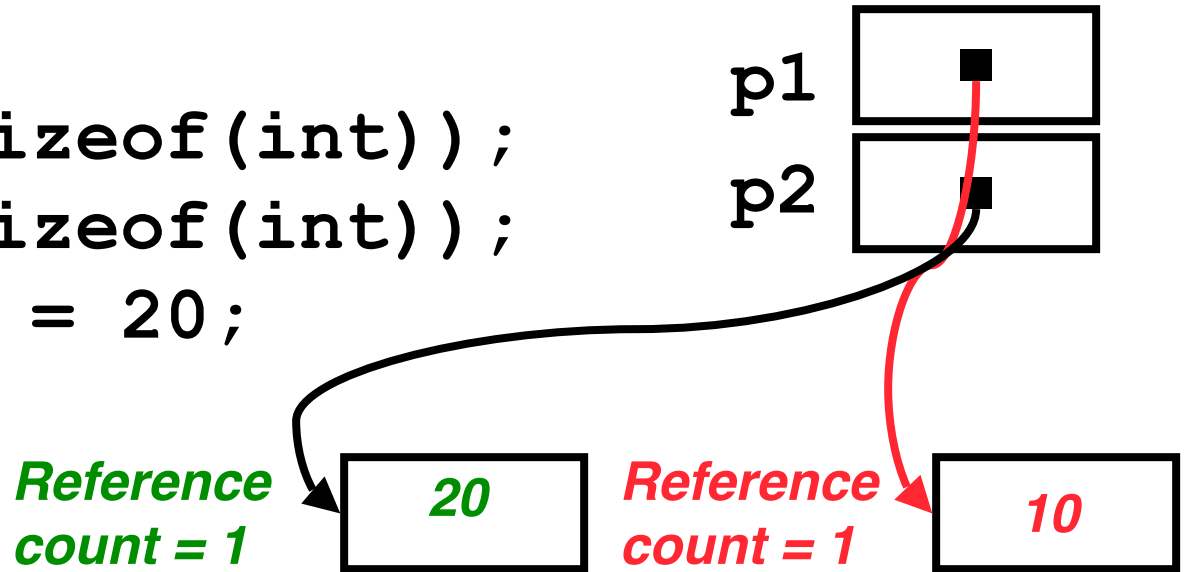
- **For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.**
- **When the count reaches 0, reclaim.**
- **Simple assignment statements can result in a lot of work, since may update reference counts of many items**



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
 - When the count reaches 0, reclaim.

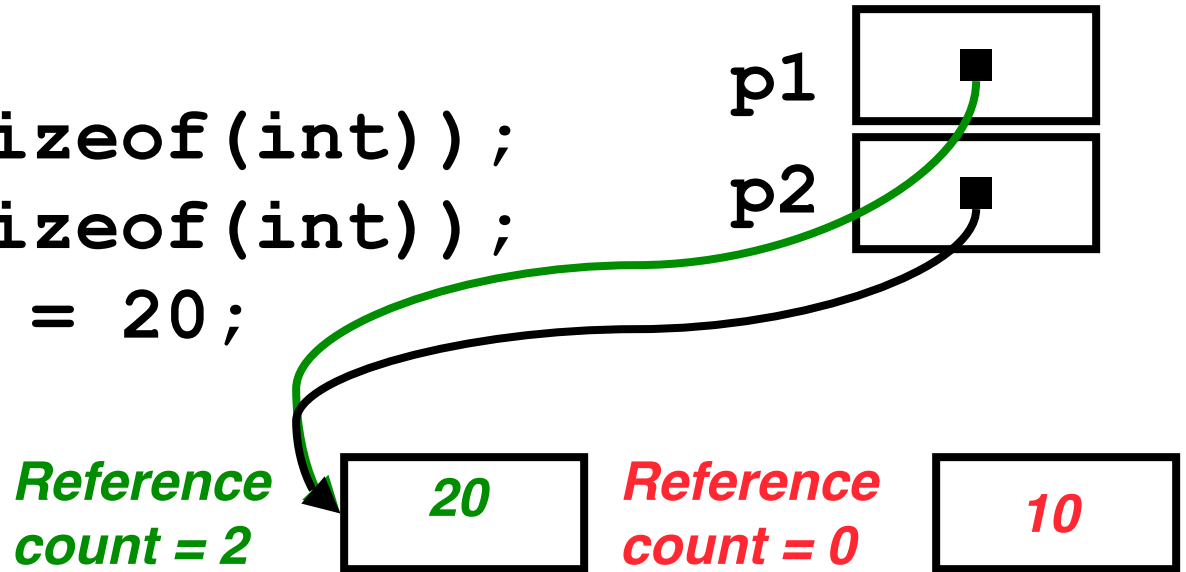
```
int *p1, *p2;  
p1 = malloc(sizeof(int));  
p2 = malloc(sizeof(int));  
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
 - When the count reaches 0, reclaim.

```
int *p1, *p2;  
p1 = malloc(sizeof(int));  
p2 = malloc(sizeof(int));  
*p1 = 10; *p2 = 20;  
p1 = p2;
```



Reference Counting (p1, p2 are pointers)

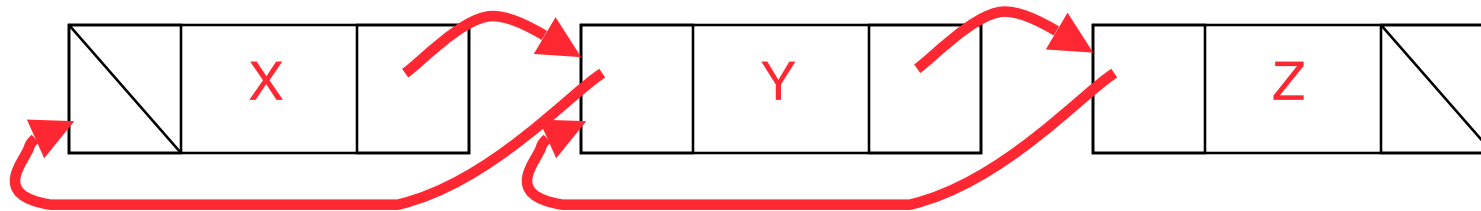
`p1 = p2;`

- Increment reference count for p2
- If p1 held a valid value, decrement its reference count
- If the reference count for p1 is now 0, reclaim the storage it points to.
 - If the storage pointed to by p1 held other pointers, decrement all of their reference counts, and so on...
- Must also decrement reference count when local variables cease to exist.



Reference Counting Flaws

- Extra overhead added to assignments, as well as ending a block of code.
- Does not work for circular structures!
 - E.g., doubly linked list:



Scheme 2: Mark and Sweep Garbage Col.

- **Keep allocating new memory until memory is exhausted, then try to find unused memory.**
- **Consider objects in heap a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.**
 - **Edge from A to B \Rightarrow A stores pointer to B**
- **Can start with the root set, perform a graph traversal, find all usable memory!**
- **2 Phases:**
 - 1. Mark used nodes**
 - 2. Sweep free ones, returning list of free nodes**



Mark and Sweep

- Graph traversal is relatively easy to implement recursively

```
void traverse(struct graph_node *node) {  
    /* visit this node */  
    foreach child in node->children {  
        traverse(child);  
    }  
}
```

- But with recursion, state is stored on the execution stack.
 - Garbage collection is invoked when not much memory left
- As before, we could traverse in constant space (by reversing pointers)



Scheme 3: Copying Garbage Collection

- **Divide memory into two spaces, only one in use at any time.**
- **When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.**
 - **Only reachable objects are copied!**
- **Use “forwarding pointers” to keep consistency**
 - **Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied (see bonus slides)**



Peer Instruction – Pros and Cons of fits

- A. The con of **first-fit** is that it results in many **small blocks** at the beginning of the free list
- B. The con of **next-fit** is it is **slower than first-fit**, since it takes longer in steady state to find a match
- C. The con of **best-fit** is that it **leaves lots of tiny blocks**

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT



Peer Instruction

- A. Of {K&R, Slab, Buddy}, there is no best (it depends on the problem).
- B. Since automatic garbage collection can occur any time, it is **more difficult to measure the execution time** of a Java program vs. a C program.
- C. We don't have automatic garbage collection in C because of **efficiency**.

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT



“And in Conclusion...”

- **Several techniques for managing heap via malloc and free: best-, first-, next-fit**
 - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
 - Each technique has strengths and weaknesses, none is definitively best
- **Automatic memory management relieves programmer from managing memory.**
 - All require help from language and compiler
 - **Reference Count:** not for circular structures
 - **Mark and Sweep:** complicated and slow, works
 - **Copying:** Divides memory to copy good stuff



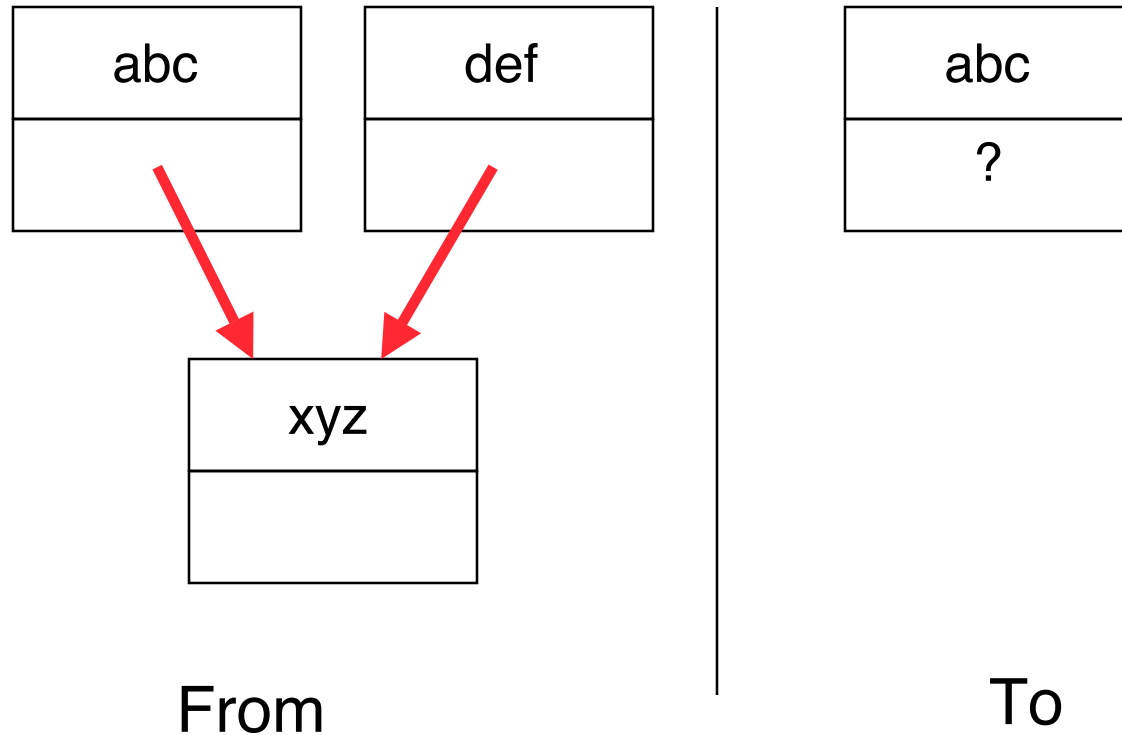
Bonus slides

- **These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.**
- **The slides will appear in the order they would have in the normal presentation**

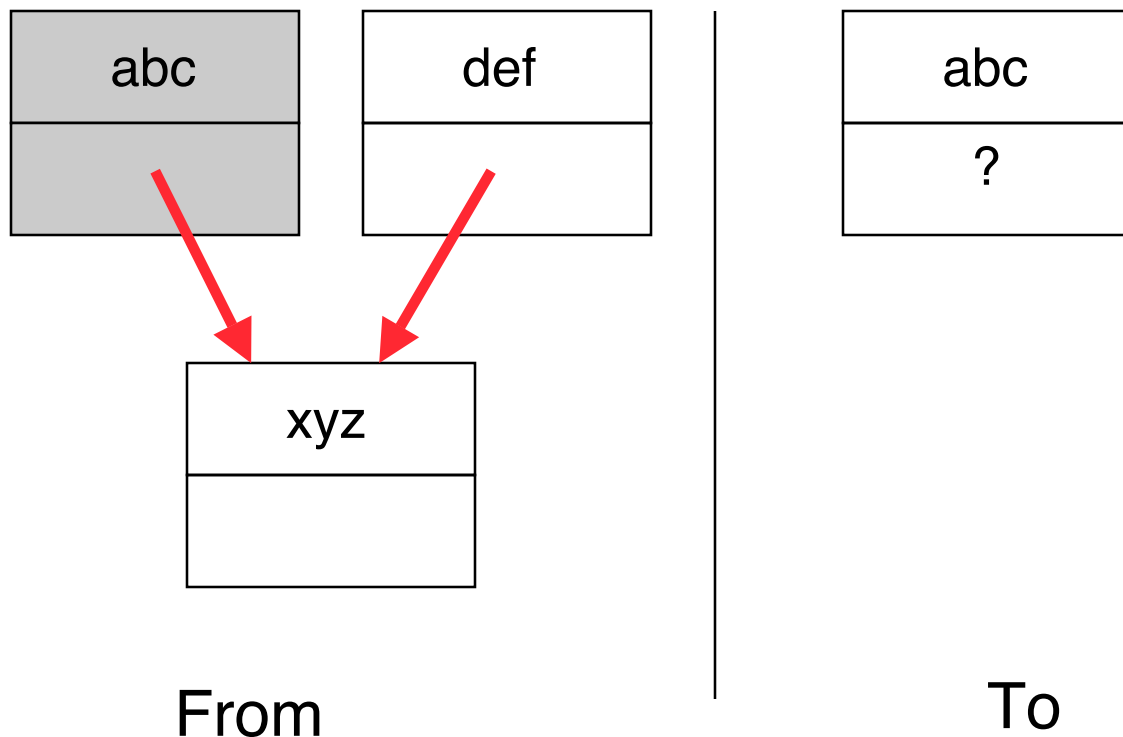
Bonus



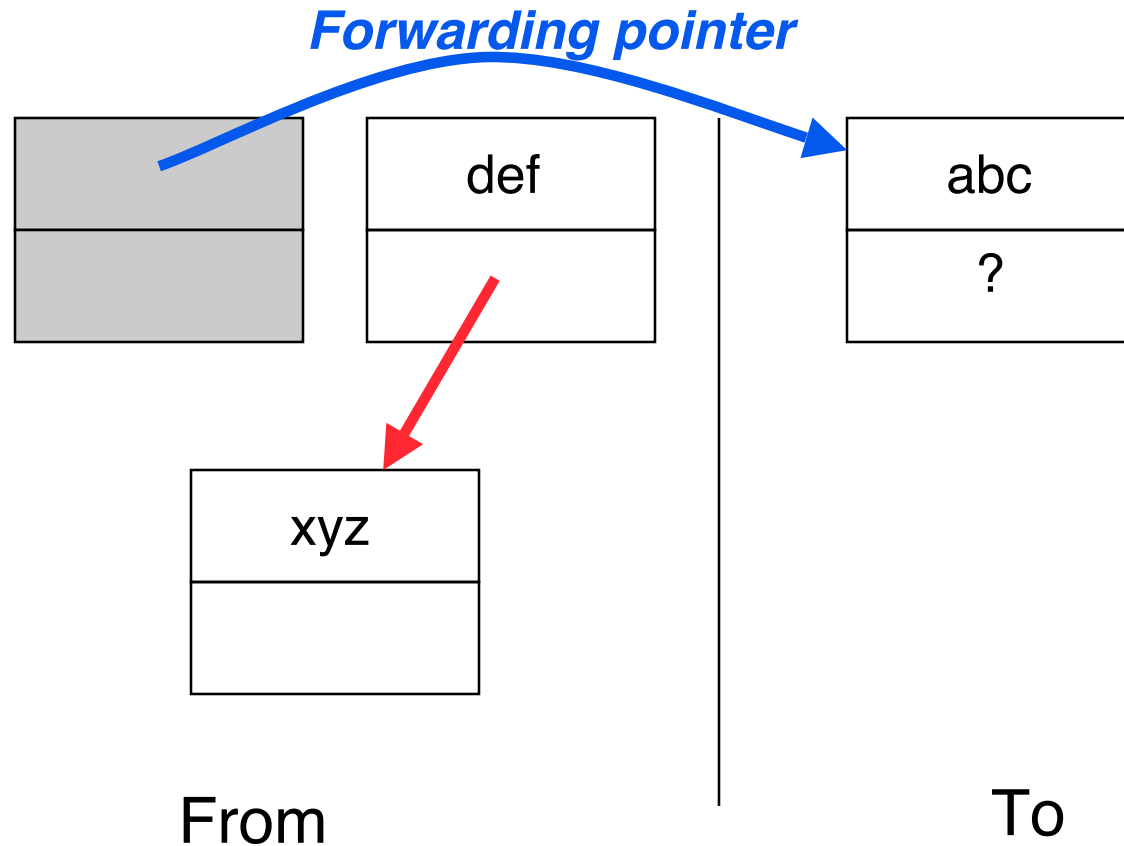
Forwarding Pointers: 1st copy “abc”



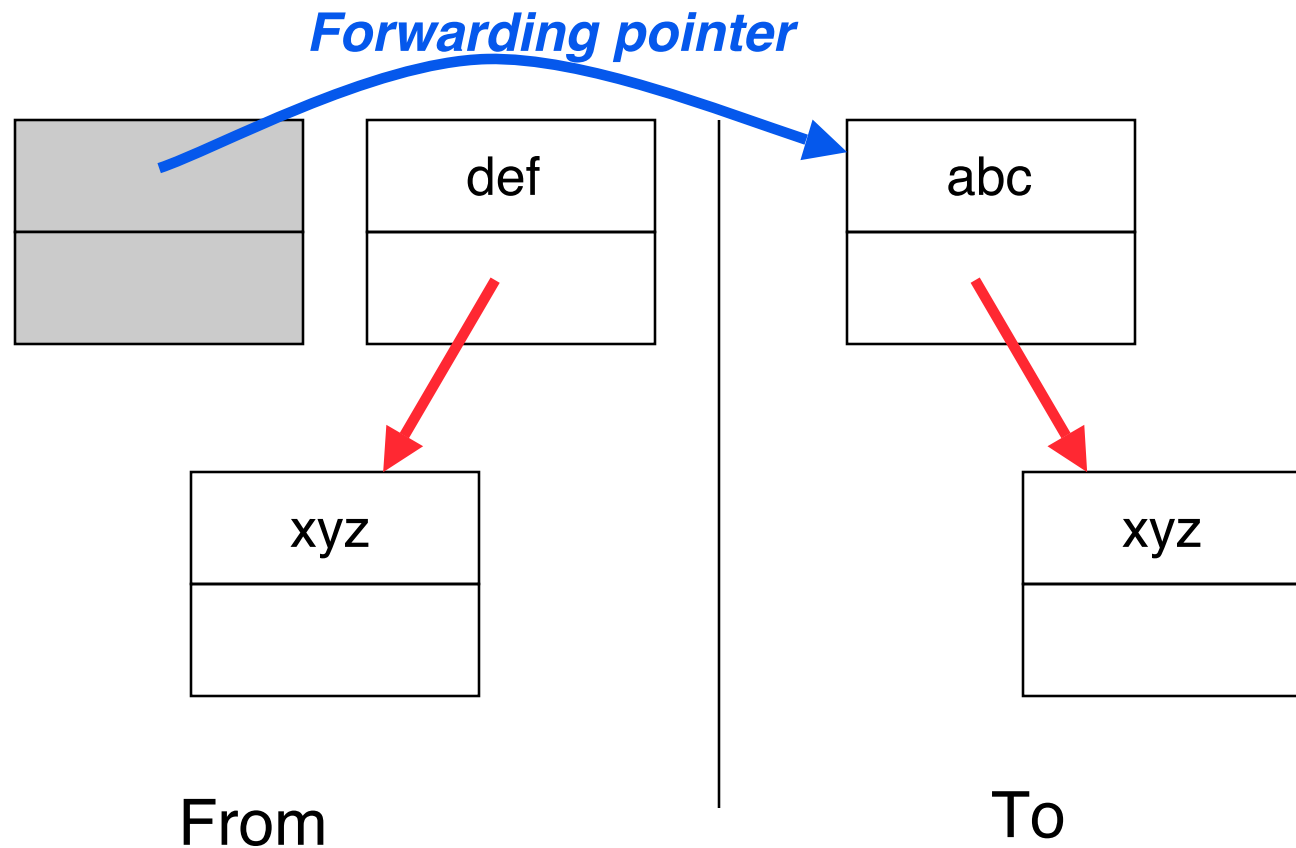
Forwarding Pointers: leave ptr to new abc



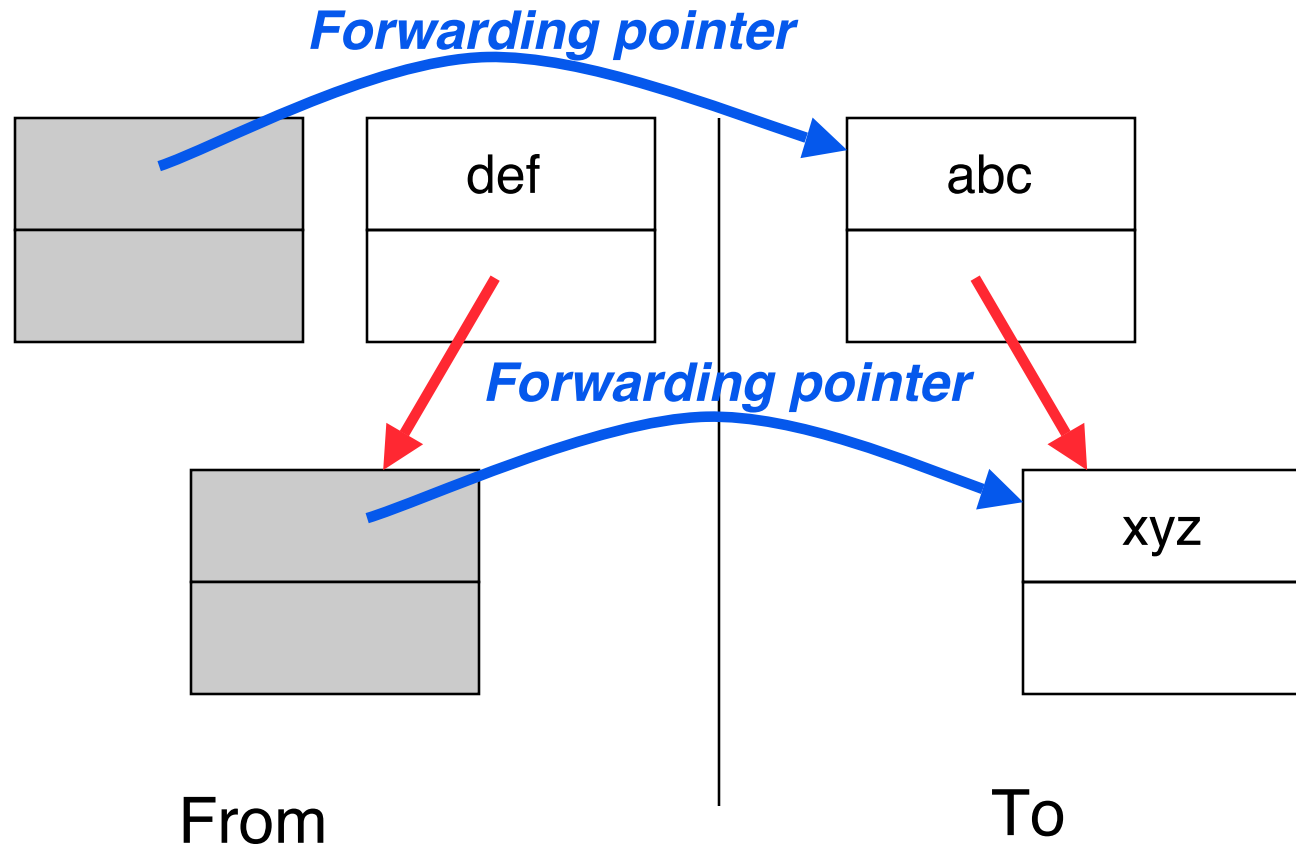
Forwarding Pointers : now copy “xyz”



Forwarding Pointers: leave ptr to new xyz



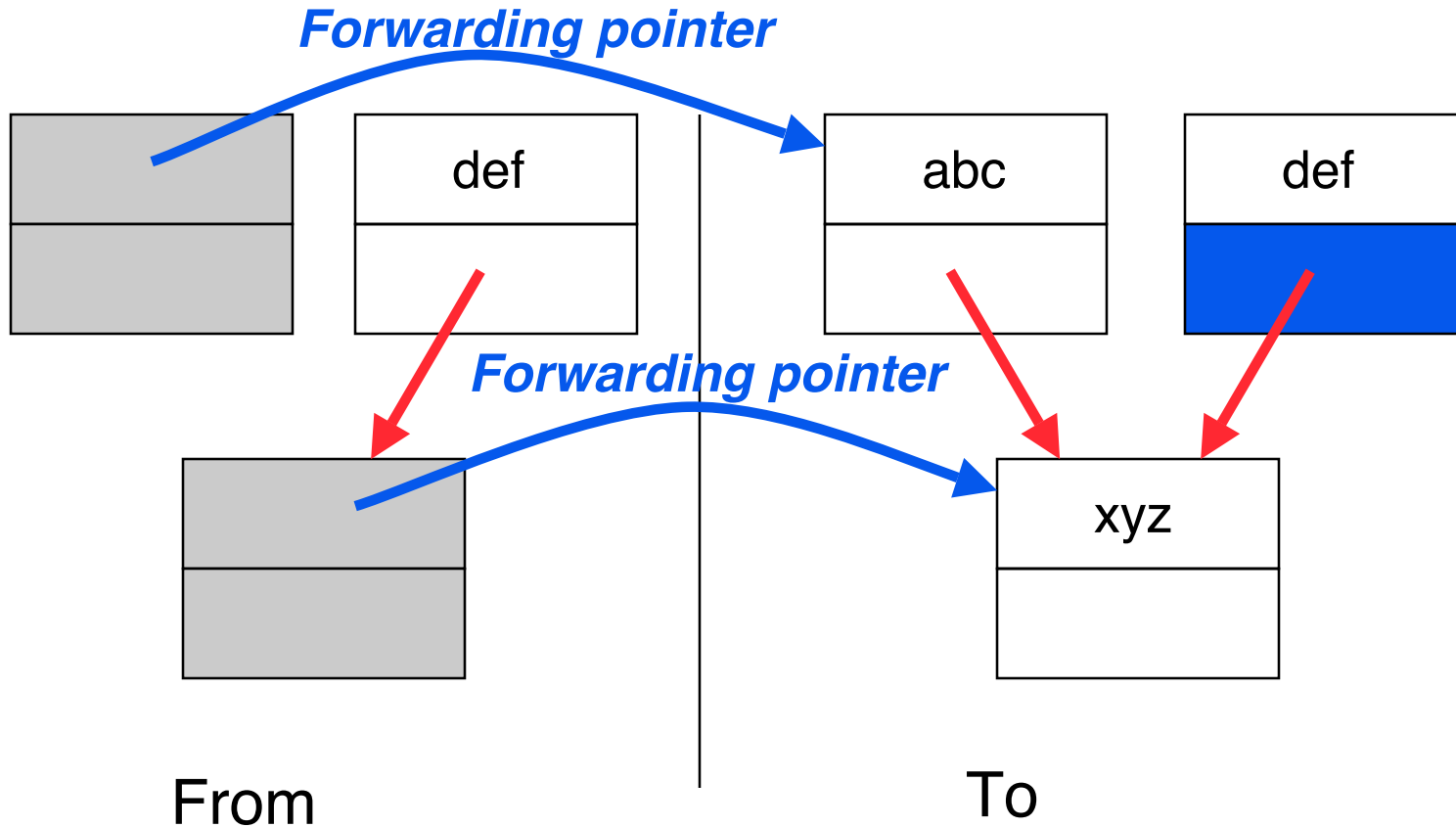
Forwarding Pointers: now copy “def”



Since xyz was already copied, def uses xyz's forwarding pointer to find its new location



Forwarding Pointers



Since xyz was already copied, def uses xyz's forwarding pointer to find its new location

